

# Appendix C

## Obtaining Configuration Information

Obtaining ISA Configuration Information .....	C-1
Obtaining EISA Configuration Information .....	C-2
Getting the Real Mode Workspace .....	C-2
Locking the Memory .....	C-3
Making a Real Mode BIOS Call .....	C-3
Accessing the Configuration Information .....	C-6
Unlocking the Memory .....	C-6
Obtaining MCA Configuration Information .....	C-7
Scanning Slots for the Adapter's ID .....	C-7
Determining the Slot to Use .....	C-8
Accessing the Configuration Information .....	C-10
Deselecting the Card .....	C-10
Registering the Configuration Information .....	C-11



## Obtaining ISA Configuration Information

The ISA BUS does not provide a standardized way to obtain hardware configuration information. Individual slots cannot be queried to determine the adapter type which is installed, nor can adapters be enabled or disabled in a uniform way. Drivers must utilize the parameters passed from the *ParseDriverParameters* call (parameters supplied in the load command line), then verify that the hardware is present and operational as specified. Some adapters may allow all other parameters to be obtained by I/O commands once a primary I/O port is identified, but drivers will still have to interpret the fields obtained this way.

## Obtaining EISA Configuration Information

NetWare device drivers on DOS-based servers can obtain EISA information by the following procedure:

- get real mode workspace
- lock the memory
- make the EISA BIOS call
- access the configuration information
- unlock the memory

Each step in this procedure is detailed in the remainder of this section.

### Getting the Real Mode Workspace

In order to read EISA configuration information, the driver must allocate a block of memory addressable in both real and protected mode. The EISA machine BIOS uses this block of memory to pass configuration information to the driver.

To obtain this block of memory, the driver must use the operating system routine *GetRealModeWorkSpace*. Before doing this, however, the driver must allocate five storage locations. These locations can be reserved in the driver's data area as follows:

<i>WorkSpaceSize</i>	dd	0	;block size
<i>WorkSpaceRealModeOffset</i>	dw	0	;block offset
<i>WorkSpaceRealModeSegment</i>	dw	0	;block segment
<i>WorkSpaceProtectedModeAddress</i>	dd	0	;block address
<i>WorkSpaceSemaphore</i>	dd	0	;block semaphore

#### *WorkSpaceSize*

Size of the block of memory (in bytes).

#### *WorkSpaceRealModeOffset*

Offset of the real mode memory address of the block.

#### *WorkSpaceRealModeSegment*

Segment of the real mode memory address of the block.

#### *WorkSpaceProtectedModeAddress*

Protected mode logical address of the block.

#### *WorkSpaceSemaphore*

Pointer to a semaphore structure. The driver uses the semaphore to "lock" the memory for its exclusive use while reading the EISA configuration information.

*Note:* Another way to reserve this space is to allocate it *on the stack*, since it will only be used for this process, and can be discarded when finished. This is a very efficient way to deal with this particular requirement, since no *Alloc* call or resource tag, etc. is required.

The driver passes the addresses of the storage locations on the stack when calling *GetRealModeWorkSpace*. This procedure provides the driver with access to the special block of memory by filling in the storage locations with the needed values. On return, the driver must clean up the stack.

push	OFFSET	WorkSpaceSize
push	OFFSET	WorkSpaceRealModeOffset
push	OFFSET	WorkSpaceRealModeSegment
push	OFFSET	WorkSpaceProtectedModeAddress
push	OFFSET	WorkSpaceSemaphore
call		GetRealModeWorkSpace
add		esp, 5*4

## Locking the Memory

During the EISA configuration read operation, the driver must have exclusive use of the special memory block. It must "lock" the memory block by calling the *CPSemaphore* function, as shown in the following example:

push	WorkSpaceSemaphore	;load semaphore
call	CPSemaphore	;lock work space
add	esp, 1*4	;adjust stack

## Making a Real Mode BIOS Call

In order for the EISA machine BIOS to pass the configuration data for the selected physical card back to the driver, the driver must make a real mode call to the EISA BIOS. The driver must allocate memory for two structures, *InputParms* and *OutputParms*.

*Note:* These structures can also be allocated on the stack.

```
InputStructure      struc
    IAXRegister     dw      ?
    IBXRegister     dw      ?
    ICXRegister     dw      ?
    IDXRegister     dw      ?
    IBPRegister     dw      ?
    ISIRegister     dw      ?
    IDIRegister     dw      ?
    IDSRegister     dw      ?
    IESRegister     dw      ?
    IIntNumber      dw      ?
InputStructure      ends

OutputStructure     struc
    OAXRegister     dw      ?
    OBXRegister     dw      ?
    OCXRegister     dw      ?
    ODXRegister     dw      ?
    OBPRegister     dw      ?
    OSIRegister     dw      ?
    ODIRegister     dw      ?
    ODSRegister     dw      ?
    OESRegister     dw      ?
    Oflags          dw      ?
OutputStructure     ends

InputParms          InputStructure  <>
OutputParms         OutputStructure <>
```

Before making the *DoRealModeInterrupt* call, the driver must fill in the *InputParms* structure as follows:

*IAXRegister*

The read configuration parameter 0D801h (See the EISA BIOS call information supplied by the EISA computer manufacturer).

*ICXRegister*

The adapter slot and block of configuration data to read. CL is the slot and CH is the block.

*IDSRegister*

The real mode segment address of where to put the block of data. This value was returned in the *WorkSpaceRealModeSegment* variable by *GetRealModeWorkSpace*.

*ISIRegister*

The real mode memory offset of where to put the block of data. This value was returned in the *WorkSpaceRealModeOffset* variable by *GetRealModeWorkSpace*.

*IIntNumber*

The interrupt number. In this example, it is interrupt 15h.

After filling out the *InputParms* structure, the driver pushes the offsets of *InputParms* and *OutputParms* and then calls *DoRealModeInterrupt*, as shown below.

```
push    OFFSET OutputParms    ;output registers
push    OFFSET InputParms    ;input registers
call    DoRealModeInterrupt  ;perform real mode int
add     esp, 2*4              ;restore stack
```

## Checking for Errors

To determine if *DoRealModeInterrupt* executed without errors, compare EAX to 0. If the value is 0, the routine executed successfully.

The driver must also detect errors the BIOS routine may have had. It does this by checking the *OAXRegister* field in the *OutputParms* structure. To determine if the BIOS routine executed without errors, compare the *OAXRegister* field to 0. If the value is 0, the routine executed successfully. A sample of the error checking code required follows:

```
or      eax, eax              ;successful?
jnz     IntNotValidErrorExit  ;jmp if OS error
cmp     BYTE PTR OutputParms.OAXRegister+1, 0
jne     IntNotValidErrorExit  ;jmp if BIOS error
```

*Note:* The error handling routines for the above errors must unlock the block of memory by calling *CVSemaphore*.

## Accessing the Configuration Information

At this point, the driver has access to the configuration of the adapter set by the user in the EISA configuration utility. The driver accesses this information using the logical address (protected mode address) of the special memory block that was returned during the *GetRealMode-Workspace* call. A sample of typical driver processing follows:

```
mov     esi, WorkspaceProtectedModeAddress    ;data pointer
mov     cl, BYTE PTR [esi+INTERRUPTOFFSET]    ;get interrupt
mov     SaveInterrupt, cl                    ;save
```

*Note:* *INTERRUPTOFFSET* is defined in the EISA spec.

Each configuration block contains different information (interrupts, memory, etc.). If the first block read does not contain the appropriate information, keep reading blocks by incrementing CH in the *InputParms* structure and calling *DoRealModeInterrupt* again. Read blocks until the information is obtained or until INT 15h returns an 81h in AH of the *OutputParms* structure.

## Unlocking the Memory

Finally, the driver must unlock the special memory block that the EISA configuration data is located in. This is accomplished by making a call to the *CVSemaphore* function, as indicated in the following example:

```
push     WorkspaceSemaphore                ;pass semaphore
call     CVSemaphore                      ;unlock workspace
add      esp, 1*4                         ;clean up stack
```

## Obtaining MCA Configuration Information

This section describes the procedure for obtaining hardware configuration information for a Micro Channel MSL adapter. This information is used by the driver's initialization procedure to initialize the hardware and to register the adapter with the NetWare OS.

The procedure for obtaining MCA configuration information is outlined below:

- scan the slots for the MSL ID
- determine which slot to use
- access the configuration information
- deselect the card
- register the configuration with the OS

The remainder of this section describes this procedure in more detail. Each step is illustrated with sample code.

### Scanning Slots for the Adapter's ID

The first step in obtaining the MSL adapter's configuration is to scan through all Micro Channel adapter slots, searching for adapter IDs supported by the MSL driver. The number of adapters with matching IDs and the corresponding slot numbers are stored in a slot options table for a later step. If no adapters with matching IDs are found, the MSL driver should display an error message stating that no adapters were found, and exit back to the OS with a completion code of non-zero.

The following definitions are used in the example on the facing page. For a more detailed explanation of the ports in the Micro Channel architecture, refer to the IBM Personal System/2 Hardware Interface Technical Reference.

```
SlotSelectRegister    equ    96h
POS0                  equ    100h
POS1                  equ    101h
POS2                  equ    102h
```

The driver must fill in a slot options table for the *AdapterOptions-Structure*. This structure is required by the *ParseDriverParameters* routine. The slot options table has the following format:

```
SlotCount    dd    0           ;number of valid slots
SlotList     dd    8 dup (0)   ;up to 8 slots possible
```

The address of the slot options table is placed in the appropriate field of the *AdapterOptionStructure* for the *ParseDriverParameters* routine.

```
AdapterOptions  AdapterOptionStructure  <SlotCount>
```

```
    xor     esi, esi                ;Init card counter
    mov     cl, 7                  ;3rd bit must be set
                                    ;Start with slot 0

ScanSlots:

    inc     cl                     ;next slot
    cmp     cl, 10h                ;are we done?
    jz      short DoneScanningSlots ;jump if so

    mov     al, cl
    out     SlotSelectRegister, al ;select card slot

    mov     dx, POS0               ;get high byte of
    in      al, dx                 ; signature
    mov     ah, al
    mov     dx, POS1               ;get low byte of
    in      al, dx                 ; signature
    xchg    al, ah

    cmp     ax, NE232ID            ;Is it our card?
    jne     ScanSlots              ;jump if not

    mov     dx, POS2               ;get config port
    in      al, dx                 ;get configuration

    test    al, 01                 ;Is card enabled?
    jz      ScanSlots              ;jump if not

    movzx   eax, cl                ;get slot number
    btr     eax, 3                 ;reset bit 3
    inc     eax                    ;slots are 1 relative
    mov     SlotList [esi*4], eax  ;put slot # in table
    inc     esi                    ;bump board count

    jmp     ScanSlots              ;Keep looking

DoneScanningSlots:

    xor     al, al
    out     SlotSelectRegister, al ;De-select card
    or      esi, esi               ;Any boards found?
    jz      NoSlotsWithMyBoard     ;jump if not

    mov     SlotCount, esi         ;Record number of boards
```

## Determining the Slot to Use

The next step is for the MSL driver (by means of the OS routine *ParseDriverParameters*) to get the slot of the adapter the MSL driver should interrogate to get the hardware configuration. Several parameters should be passed to the OS routine *ParseDriverParameters*:

- the load module *ScreenHandle*  
(see the driver initialization specification for more detail)
- the load *CommandLine* pointer  
(see the driver initialization specification for more detail)

- a bit map indicating that the slot needs to be parsed (see *NeedsBitMap* under the *ParseDriverParameters* description)
- the address to any frame description (null in the case of all MSL drivers)
- the address of any configuration limitations (null in the case of all MSL drivers)
- the address of the *AdapterOptions* structure (containing the address of the slot options table)
- the address of the driver configuration table (null in the case of all MSL drivers)
- the address of the *IOConfigurationStructure*

The sample below shows how you might parse which board in which slot to use.

Example of the parse slot number code:

```
DriverInitialize proc
    CPush
    mov     ebp, esp
    pushfd
    cli
    .
    .
    .

    push    [ebp + Parm1]           ;Screen ID
    push    [ebp + Parm2]           ;Command line
    push    NeedsIOSlotBit          ;Parse for slot
    push    0                       ;No FrameTypeDesc
    push    0                       ;No ConfigLimits
    push    OFFSET AdapterOptions
    push    0                       ;No ConfigTable
    push    OFFSET IOConfiguration ;IOConfigStruct

    call    ParseDriverParameters
    add     esp, 8 * 4

    or      eax, eax
    jnz     ErrorParsingIOParameters
DriverInitialize endp
```

Note that even if the MSL driver finds a supported adapter in a slot, another driver could be in control of the adapter. The OS routine *ParseDriverParameters* would then return a non-zero value (CCode in EAX) telling the MSL driver not to use any adapter. The MSL driver would then display a message indicating that no adapter was selected, and exit to the OS.

If *ParseDriverParameters* returns successfully, the OS parsed slot value will be placed in the *IOConfiguration* structure. Based on that value, the MSL driver can now interrogate the selected adapter for its hardware configuration as described in the following section.

## Accessing the Configuration Information

The example below illustrates these steps. The following is the code from the HNE232 driver. All adapters will have specific information for each adapter in the POS option select data bytes.

```
;Use the slot chosen to determine Memory base, I/O base and interrupt

movzx  eax, IOConfiguration.CSlot          ;Get slot that we are to use
dec    eax                                ;Make it zero-relative
bts    eax, 3                             ;PS/2 needs bit 3 set
out    SlotSelectRegister, al             ;Select the card slot

mov    dx, POS2                           ;Get config port

in     al, dx                             ;Get configuration
movzx  edi, al                            ;Copy config
and    edi, 0eh                           ;Get bits 3-1, use as index

shr    edi, 1                             ;Adjust for table index
dec    edi                                ;Option 0 is no option
shl    edi, 2                             ;Adjust for table index
mov    eax, RAMTable[edi]                  ;Get memory location used
mov    IOConfiguration.CMemoryDecode0, eax
mov    IOConfiguration.CMemoryLength0, 800h ;Store length of memory

shr    edi, 1                             ;Adjust for table index
mov    cx, IOTable[edi]                   ;Look up IO address
mov    IOConfiguration.CIOPortsAndLengths, cx

in     al, dx                             ;Get configuration again
movzx  edi, al                            ;Copy configuration
shr    edi, 4                             ;Get IRQ setting
and    edi, 7                             ;Only keep the lower 2 bits

mov    al, IRQTable[edi]                   ;Look up interrupt level
mov    IOConfiguration.CInterrupt, al      ;Save our int value

xor    al, al                             ;De-select card
out    SlotSelectRegister, al

mov    IOConfiguration.CInterrupt + 1, -1
mov    WORD PTR IOConfiguration.CDMAUsage, -1
```

## Deselecting the Card

Remember to de-select the card when finished reading the hardware configuration information (shown in the example above).

## Registering the Configuration Information

The final step registers the hardware configuration with the NetWare operating system.

```
;Register Hardware Options to see if any conflicts  
  
    push    0  
    push    OFFSET IOConfiguration  
    call    RegisterHardwareOptions  
    add     esp, 2 * 4  
  
    or      eax, eax  
    jnz     ErrorRegisteringHardwareOptions
```

If the OS rejects the registration of the hardware configuration, the MSL driver must terminate the initialization process, and display a message indicating rejection of the hardware configuration. The MSL driver initialization routine would then set a non-zero return code and exit to the OS.

See the driver template in Appendix E for more details on how to handle errors during MSL driver initialization.

